caBench-To-Bedside Admin Module v1.0

Design Document

Document Change History

Version Number	Date	Contributor	Description
V1.0 draft	June 21, 2008	Washington University/Persistent Systems	Draft document
V1.0	July 22, 2008	Washington University/Persistent Systems	Reviewed and updated for caB2B v2.0 release

July 21, 2008

Model caBIG™ Open Source Software License
v.2
Release Date: January 7, 2008

This caBIG™ Software License (the "License") is between caBIG™ Participant and You. "You (or "Your") shall mean a person or an entity, and all other entities that control, are controlled by, or are under common control with the entity. "Control" for purposes of this definition means (i) the direct or indirect power to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

License. Provided that You agree to the conditions described below, caBIG™ Participant grants You a non-exclusive, worldwide, perpetual, fully-paid-up, no-charge, irrevocable, transferable and royalty-free right and license in its rights in the caBIG™ Software, including any copyright or patent rights therein, to (i) use, install, disclose, access, operate, execute, reproduce, copy, modify, translate, market, publicly display, publicly perform, and prepare derivative works of the caBIG™ Software in any manner and for any purpose, and to have or permit others to do so; (ii) make, have made, use, practice, sell, and offer for sale, import, and/or otherwise dispose of caBIG™ Software (or portions thereof); (iii) distribute and have distributed to and by third parties the caBIG™ Software and any modifications and derivative works thereof; and (iv) sublicense the foregoing rights set out in (i), (ii) and (iii) to third parties, including the right to license such rights to further third parties. For sake of clarity, and not by way of limitation, caBIG™ Participant shall have no right of accounting or right of payment from You or Your sublicensees for the rights granted under this License. This License is granted at no charge to You. Your downloading, copying, modifying, displaying, distributing or use of caBIG™ Software constitutes acceptance of all of the terms and conditions of this Agreement. If you do not agree to such terms and conditions, you have no right to download, copy, modify, display, distribute or use the caBIG™ Software.

Your redistributions of the source code for the caBIG™ Software must retain the above copyright notice, this list of conditions and the disclaimer and limitation of liability of Article 6 below. Your redistributions in object code form must reproduce the above copyright notice, this list of conditions and the disclaimer of Article 6 in the documentation and/or other materials provided with the distribution, if any.

Your end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by School of Medicine, Washington University in St. Louis." If You do not include such end-user documentation, You shall include this acknowledgment in the caBIG™ Software itself, wherever such third-party acknowledgments normally appear.

You may not use the names "School of Medicine, Washington University in St. Louis", "The National Cancer Institute", "NCI", "Cancer Bioinformatics Grid" or "caBIG™" to endorse or promote products derived from this caBIG™ Software. This License does not authorize You to use any trademarks, service marks, trade names, logos or product names of either caBIG™ Participant, NCI or caBIG™, except as required to comply with the terms of this License.

For sake of clarity, and not by way of limitation, You may incorporate this caBIG™ Software into Your proprietary programs and into any third party proprietary programs. However, if You incorporate the caBIG™ Software into third party proprietary programs, You agree that You are solely responsible for obtaining any permission from such third parties required to incorporate the caBIG™ Software into such third party proprietary programs and for informing Your sublicensees, including without limitation Your end-users, of their obligation to secure any required permissions from such third parties before incorporating the caBIG™ Software into such third party proprietary software programs. In the event that You fail to obtain such permissions, You agree to indemnify caBIG™ Participant for any claims against caBIG™ Participant by such third parties, except to the extent prohibited by law, resulting from Your failure to obtain such permissions.

For sake of clarity, and not by way of limitation, You may add Your own copyright statement to Your modifications and to the derivative works, and You may provide additional or different license terms and conditions in Your sublicenses of modifications of the caBIG™ Software, or any derivative works of the caBIG™ Software as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

THIS caBIG™ SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES (INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT AND FITNESS FOR A PARTICULAR PURPOSE) ARE DISCLAIMED. IN NO EVENT SHALL THE SCHOOL OF MEDICINE, WASHINGTON UNIVERSITY IN ST. LOUIS OR ITS AFFILIATES BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS caBIG™ SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Index

List of Figures

Chapter 1 Introduction

This document explains the design of the components and modules present in caBench-To-Bedside Administrator (caB2B-Admin) project. It provides details of different components that are being developed as a part of caB2B-Admin application.

Chapter 2 Loading Models

Overview

One of the basic requirements of caB2B is to be able to fetch models from the caDSR and populate metadata repository (MDR) from that. It will be used to build metadata based queries to fetch data from data services. In order to understand the design of load model in cab2b admin it is necessary to first understand the design and concept of MDR.

MDR stores the metadata for an UML model including its semantic annotations such as all CDEs including permissible values by decomposing the annotated UML model obtained from caDSR.

It also contains all-to-all paths between every two classes. Given the amount of information it stores, it is also possible to get all the paths between two classes across two different UML models based on semantic interoperability.

The design of MDR is the basic foundation for caB2B backend. It enables the caB2B query engine to provide the following functionalities:

- Metadata search
- Auto generation of user interface for entering predicates
- Automatic path resolution between two query predicates
- Category support
- Inter model queries based on semantic joins

APIs Used

Following APIs are used to fetch the DomainModel from caDSR.

Classes used are

```
import gov.nih.nci.cadsr.umlproject.domain.Project;
import gov.nih.nci.cagrid.cadsr.client.CaDSRServiceClient;
import gov.nih.nci.cagrid.metadata.dataservice.DomainModel;
```

To get projects from caDSR

```
String cadsrUrl="http://cagrid-service.nci.nih.gov:8080/wsrf/services/cagrid/CaDSRService";

//To get all the project details
CaDSRServiceClient client= new CaDSRServiceClient(cadsrUrl);
Project[] projects = client.findAllProjects();
```

To generate models from projects

```
//generate domain model from project
Project project = new Project();
project.setLongName("projectLongName");
DomainModel model = client.generateDomainModelForProject(project);
```

Implementation

Process of loading models

Process of loading models can be divided into two steps

- Fetching all the available models to load

  In the first step all the models available for the loading are fetched. Only the models which are not present in the local database are shown to the user for selection.

- Loading the user selected models

  In this step user selects the models to be loaded from UI. These user selected models are then loaded into the MDR.

Classes involved

The classes involved in this module are:

- *SearchDataHeader.jsp* is the starting point for the Load Model functionality where user selects the Load Model link from the popup menu list.
- *CaDSRLoadModel.java* is the struts action called by the Load Model button; this class is a controller used to control the flow of the request.
- *CaDSRLoadModelBizLogic.java* is used to get all the model names available for loading. This class uses CaDSRServiceClient to fetch all the models for loading. All these models are then filtered so as to choose only models that not already present in the local database. This class is also responsible for loading user selected models into the MDR using PathBuilder.
- *PathBuilder.java* class is responsible for loading the model into MDR
- *LoadModel.jsp* is the UI page which displays the model names and description for that model. It also allows user to select the multiple models to load.
- *LoadModel.java* is the struts action which is called by the LoadModel.jsp page.

Sequence Diagram

Fetch All Available Models

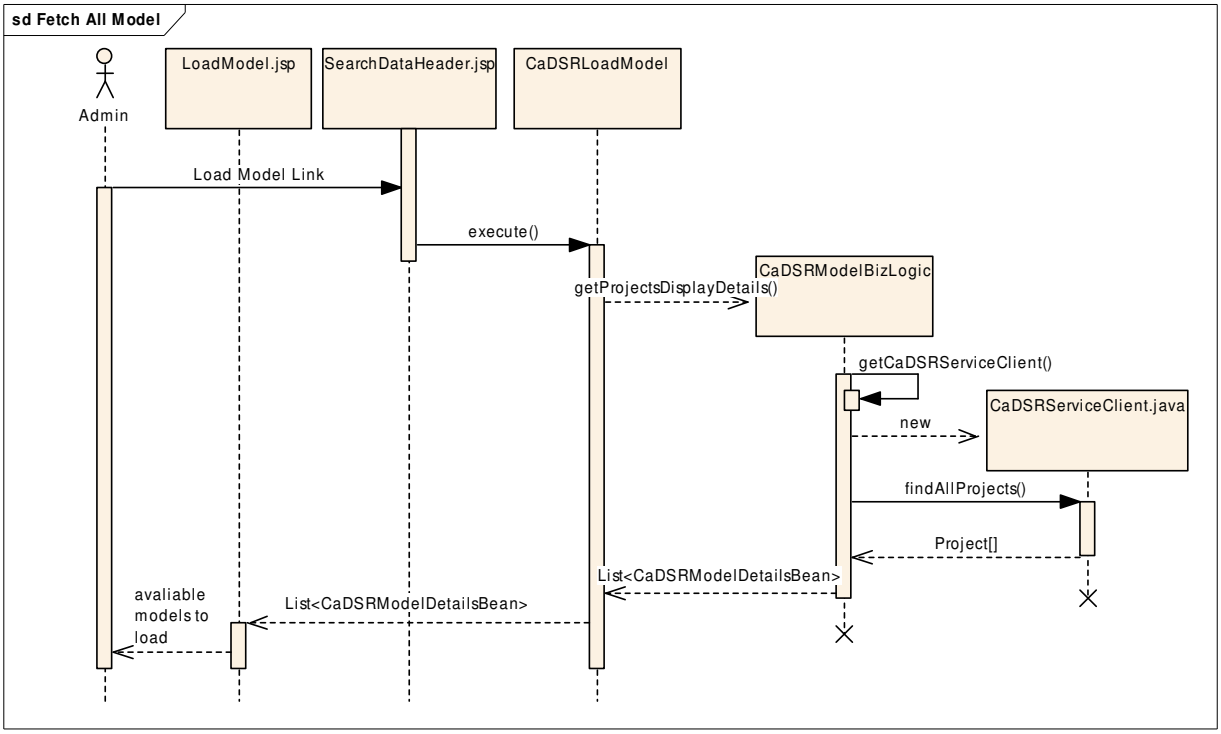

Figure 1 Sequence diagram depicting fetching all available models

Fetch Model

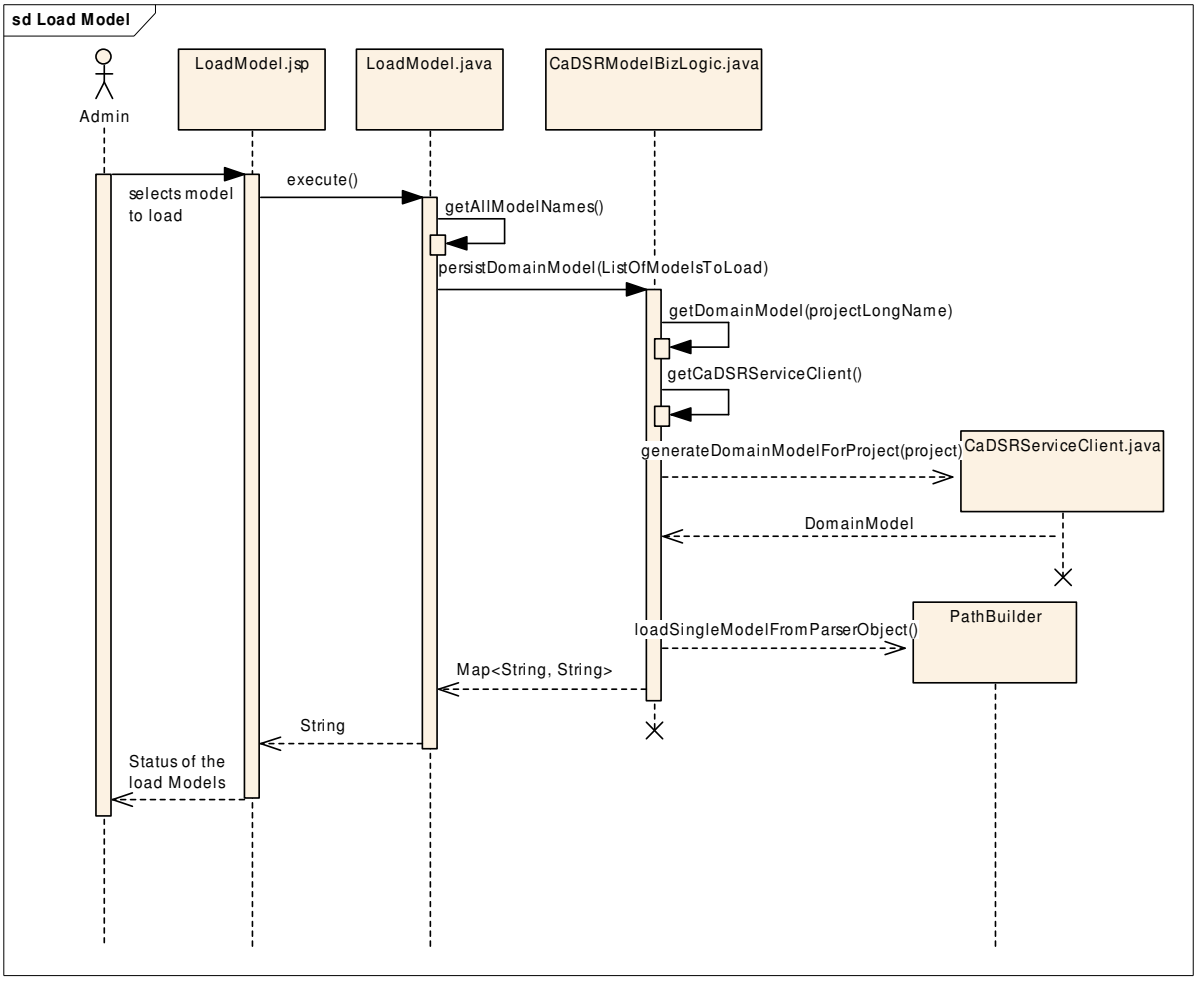

Figure 2 Sequence diagram depicting fetching selected model

Chapter 3 Configure Service Instances

Overview

Service instance configuration from admin module involves configuring the URLs of the different services for which the models are already loaded in the application. One model can have multiple service instances configured for it. This involves two stages:

- Fetching the list of all the services available for a certain model
- Saving the selected instances' URLs into database.

APIs Used

Following are the APIs used for service instance configuration functionality

Classes used are

```
import org.apache.axis.message.addressing.AttributedURI;
import org.apache.axis.message.addressing.EndpointReferenceType;
import gov.nih.nci.cagrid.discovery.client.DiscoveryClient;
import gov.nih.nci.cagrid.metadata.MetadataUtils;
import gov.nih.nci.cagrid.metadata.ServiceMetadata;
```

Fetching service instances:

```
String indexServiceUrl =
    "http://cagrid-index.nci.nih.gov:8080/wsrf/services/DefaultIndexService";
DiscoveryClient client = new DiscoveryClient(indexServiceUrl);
EndpointReferenceType[] epr = client.discoverDataServicesByDomainModel("modelName");
```

Fetching metadata for each instance:

```
AttributedURI attributeUri = endPointRef.getAddress();
ServiceMetadata metaData = MetadataUtils.getServiceMetadata(endPointRef);
String serviceDescription = metaData.getServiceDescription().toString();
String hostingCenter = metaData.getHostingResearchCenter().toString();
```

Application flow

Under Search Data menu, administrator selects “Service Instance”. The admin will be shown models currently present in MDR.

User clicks on the model name to see running services which are using that model Name of the selected model is passed to DiscoveryClient ‘s method, *discoverDataServicesByDomainModel(String modelName)*. It returns *EndPointReferenceType[]*.

Metadata for each of the *EndPointReferenceType* is fetched using methods on *MetadataUtils* class.

Using this metadata, an object of *ServiceMetadata* class is created for each of the *EndPointReferenceType*.

The *ServiceMetadata* objects are used to display hosting center’s information.

User selects the instances that are to be saved. These are saved through *saveServiceInstances(Collection<AdminServiceMetadata> serviceMetadataObjects, String userName)* in *ServiceInstanceBizLogic* class.

Database schema

The DDL of the tables will be

```
create table CAB2B_SERVICE_URL (
   URL_ID bigint not null auto_increment,
   ENTITY_GROUP_NAME text not null,
   URL text not null,
   ADMIN_DEFINED bit not null,
   primary key (URL_ID)
);
create table CAB2B_USER (
   USER_ID bigint not null auto_increment,
   NAME varchar(30) not null,
   PASSWORD varchar(30) not null,
   IS_ADMIN bit not null,
   primary key (USER_ID)
);
create table CAB2B_USER_URL_MAPPING (
   SERVICE_URL_ID bigint not null,
   USER_ID bigint not null,
   primary key (USER_ID, SERVICE_URL_ID)
);
```

Chapter 4 Directed Acyclic Graph (DAG)

Overview

Directed **A**cyclic **G**raph (DAG) is a component to be used in a web application which allows user to build and visualize graph. It is developed in Flex and is shared between following functionalities:

1. Create category
2. Define intermodel joins
3. Curate Paths

Class Diagram

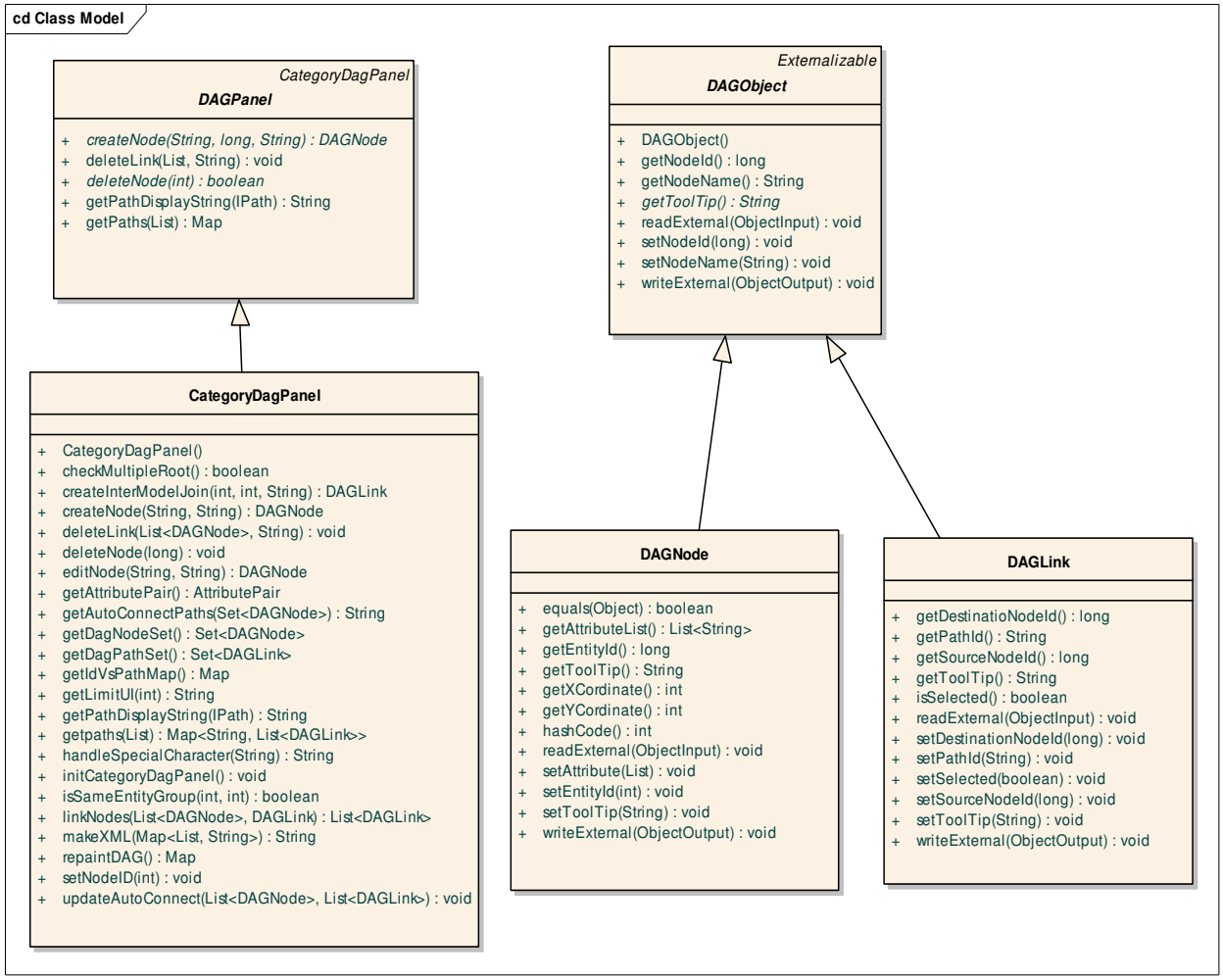

Figure 3 Class diagram of DAG

Description of classes:

- **DAGObject**: This class represents the basic structure of Directed Acyclic Graph (DAG).
- **DAGNode**: Each entity is represented by a DAGNode in DAGView. Each DAGNode has a unique Id and toolTip which shows all the attributes of an entity added in DAGView.
- **DAGLink**: It represents the link between two entities added as DAGNode.
- **DAGPanel**: This is a abstract class.
- **CategoryDagPanel**: It represents all the business methods called from Flex UI part .It is registered in remote-config.xml to create the communication between Flex (UI part) to server (java classes). This class gets called by DAG.mxml .

Chapter 5 Curate Path

Class Structure

Refer to the requirement document to understand the expected functionality of path curation. IAssociation represents a direct linkage between two classes. There are two extensions of IAssociation. IIntraModelAssociation represent the link between two classes from the same model. IInterModelAssociation represent the link between two classes from different models.

An IPath represents a navigable way between a source and a destination class. It is composted list of IAssociation. It will have at least one

ICuratedPath is defined for set of entities (classes), it is composed of one or many IPath which form a non-cyclic graph between given set of entities.

The implementing classes of these interfaces are hibernate objects. Save-Update-delete operations will be performed using DAO pattern. The interface structure will be as follows

Interface diagram

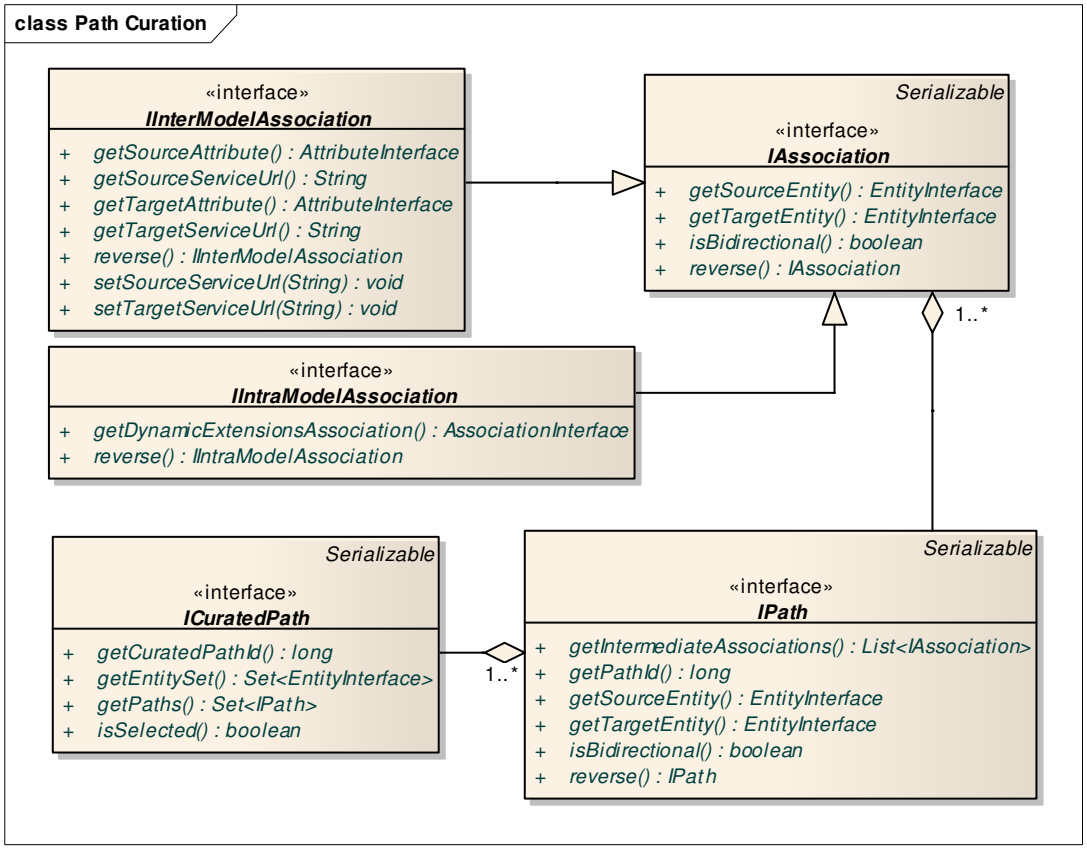

Figure 4 Path Curation Interfaces

Class Diagram

The classes involved in the path curation will be as follows

- *ModelAssociation* implements *IAssociation*
- *InterModelAssociation* implements *IInterModelAssociation* extends *ModelAssociation*
- *IntraModelAssociation* implements *IIntraModelAssociation* extends *ModelAssociation*
- *Path* implements *IPath*
- *CuratedPath* implements *ICuratedPath*

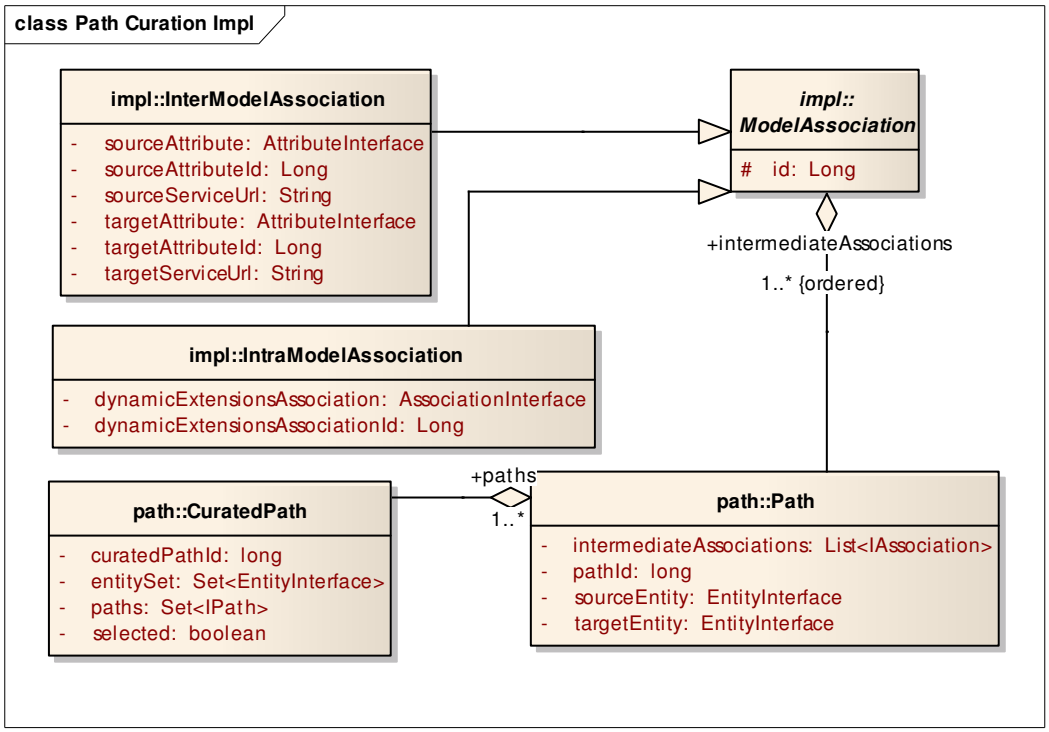

Figure 5 Path Curation Java beans

Database schema

```
/*
 * INTERMEDIATE_PATH contains ASSOCIATION(ASSOCIATION_ID)
 * connected by underscore
 */
create table PATH(
     PATH_ID           bigint         not null,
     FIRST_ENTITY_ID   bigint         null,
     INTERMEDIATE_PATH varchar(1000)  null,
     LAST_ENTITY_ID    bigint         null,
     primary key (PATH_ID),
     index INDEX1 (FIRST_ENTITY_ID,LAST_ENTITY_ID)
);

/* Possible values for ASSOCIATION_TYPE are 1 and 2
 * ASSOCIATION_TYPE = 1 represents INTER_MODEL_ASSOCIATION.
 * ASSOCIATION_TYPE = 2 represents INTRA_MODEL_ASSOCIATION.
 */
create table ASSOCIATION(
    ASSOCIATION_ID    bigint    not null,
    ASSOCIATION_TYPE  INT(8)    not null ,
    primary key (ASSOCIATION_ID)
);

create table INTER_MODEL_ASSOCIATION(
    ASSOCIATION_ID     bigint  not null references
      ASSOCIATION(ASSOCIATION_ID),
    LEFT_ENTITY_ID      bigint  not null,
    LEFT_ATTRIBUTE_ID   bigint  not null,
    RIGHT_ENTITY_ID     bigint  not null,
    RIGHT_ATTRIBUTE_ID  bigint  not null,
    primary key (ASSOCIATION_ID)
);

create table INTRA_MODEL_ASSOCIATION(
    ASSOCIATION_ID    bigint    not null references
ASSOCIATION(ASSOCIATION_ID),
    DE_ASSOCIATION_ID bigint    not null,
    primary key (ASSOCIATION_ID)
);

create table CURATED_PATH (
      curated_path_Id BIGINT,
      entity_ids VARCHAR(1000),
      selected boolean,
      primary key (curated_path_Id)
);

/*this is mapping table for many-to-many relationship between
tables PATH and CURATED_PATH */
create table CURATED_PATH_TO_PATH (
      curated_path_Id BIGINT references CURATED_PATH
(curated_path_Id),
      path_id BIGINT  references PATH (path_id),
      primary key (curated_path_Id,path_id));
```

User interface

The user interface of the path curation will be created using Flex. The communication between Flex and java will happen as shown in the diagram below.

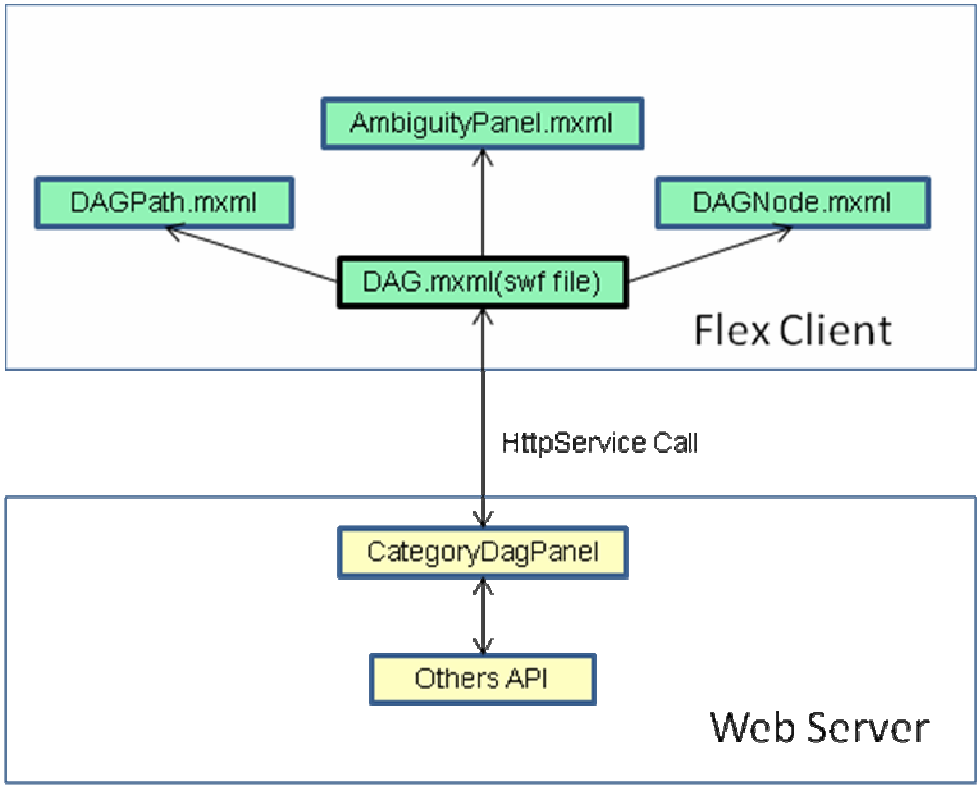

Figure 6 User Interface diagram of Curate Path

DAGObject: It is a java bean, which is base class for all the components that can be displayed in

AmbiguityPanel.mxml: Flex component to show all the paths and allows admin to choose one which will be part of curated path.

DAGNode.mxml: Flex component to represent a link between two nodes. The java representation of this is DAGNode

DAGPath.mxml: Flex component to represent a link between two nodes. The java representation of this is DAGLink

DAG.mxml: Main flex component which displays all other components. This is the class which will get called from java script. This is flex side of the component

CategoryDAGpanel: All communication from java to flex will be handled by HTTPSevice calls between DAG.mxml and CategoryDAGpanel .This extends DAGPanel and implements all the business logic.

DAGPanel: It contains all the business logic needed for path curation.

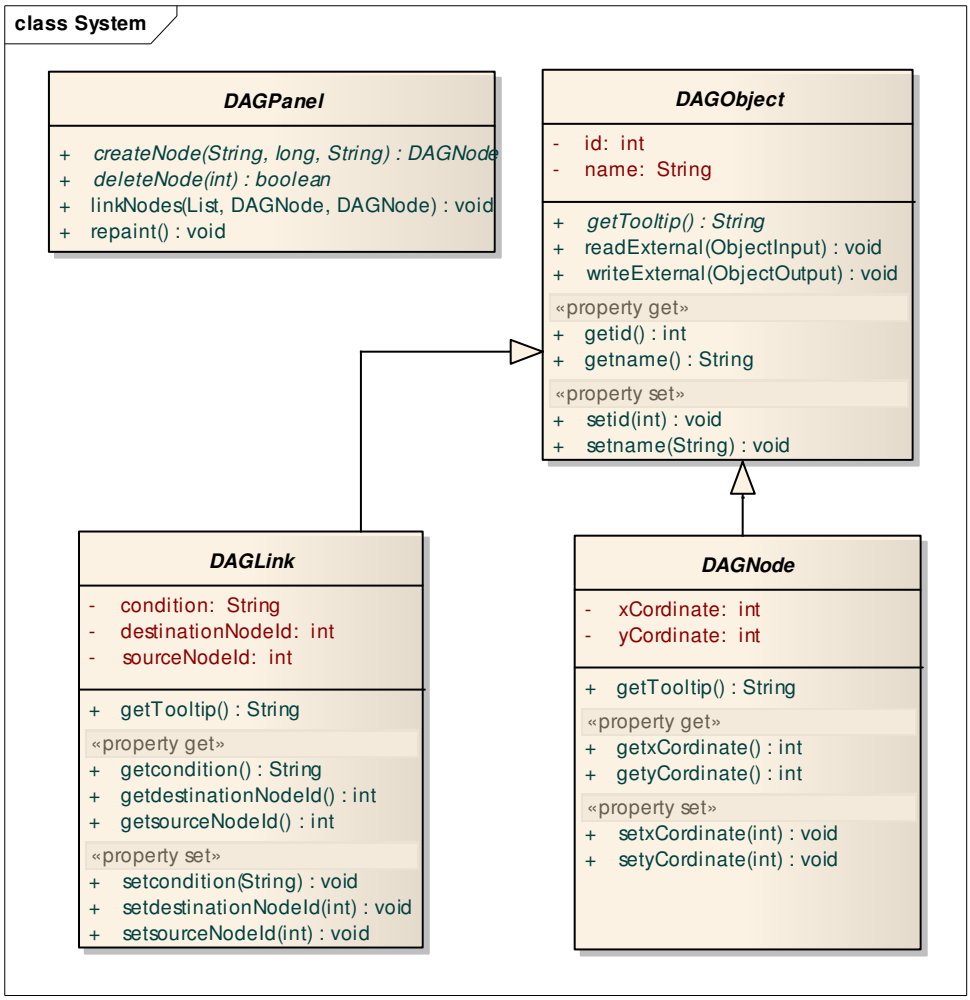

Figure 7 Class diagram of Curate Path UI

Sequence Diagram

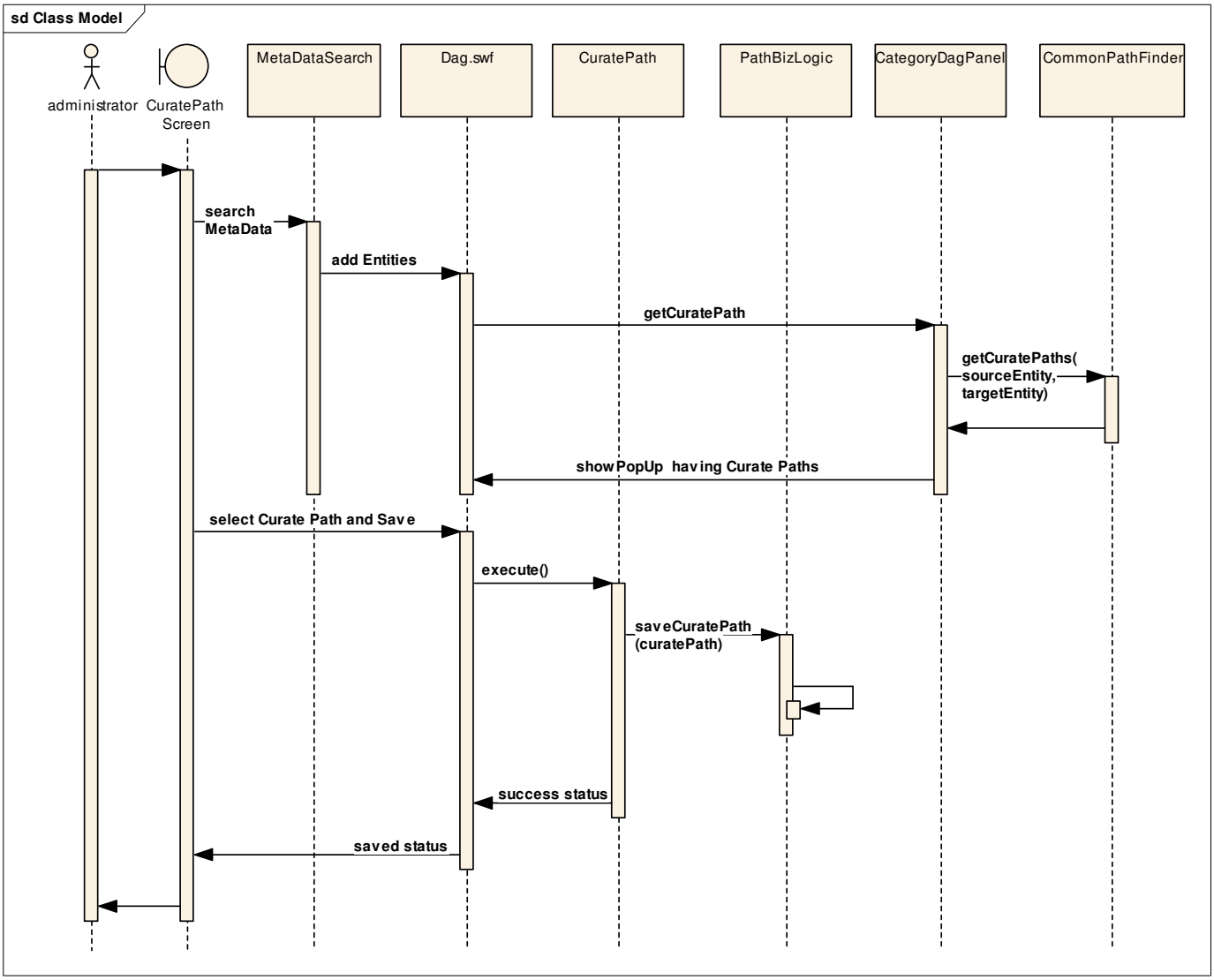

Figure 8 Sequence diagram of path curation

After searching the classes from metadata search, administrator can define curate paths among these classes. On clicking the links of searched classes will add them into the DAG, select two classes at a time and click on the manual connect button on the DAG.

At this action *DAG.mxml* calls the *getPaths()* of *CategoryDagPanel* which calls the *getCuratePaths(sourceEntity,targetEntity)* of *CommonPathFinder* .It returns the curate paths between *sourceEntity* class and *targetEntity* class.

If there are multiple paths, ambiguity resolver pops up with all paths between source class and target class. This activity is performed for each set of classes user wants to connect. Clicking of save button will save the selected path by calling *saveCuratePath(curatePath)* of *PathBizLogic*. This method will save the curated path for the selected classes.

Chapter 6 Category Creation

Overview

In Create Category module, different entities or classes can be combined together to generate a category. The attributes that are selected form the different entities will now collectively become attributes of category. This is analogous to join of SQL.

Sequence Diagrams

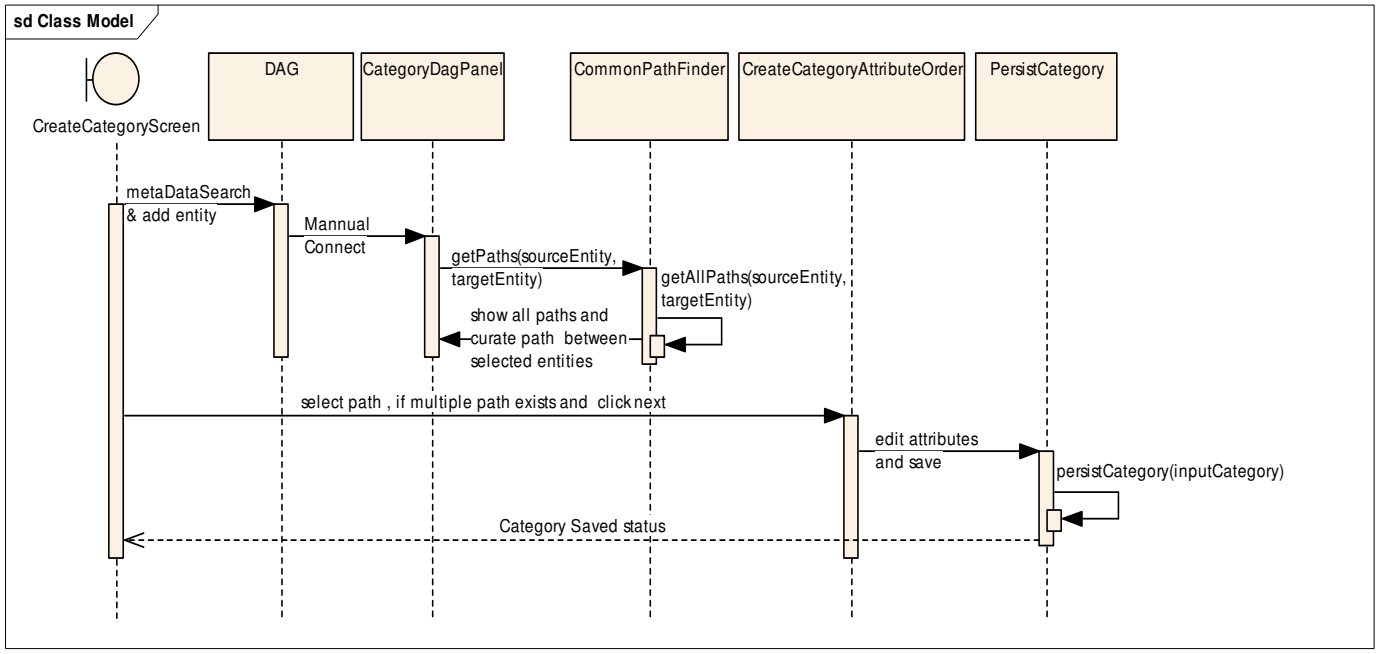

Figure 9 Category creation with Manual Connect

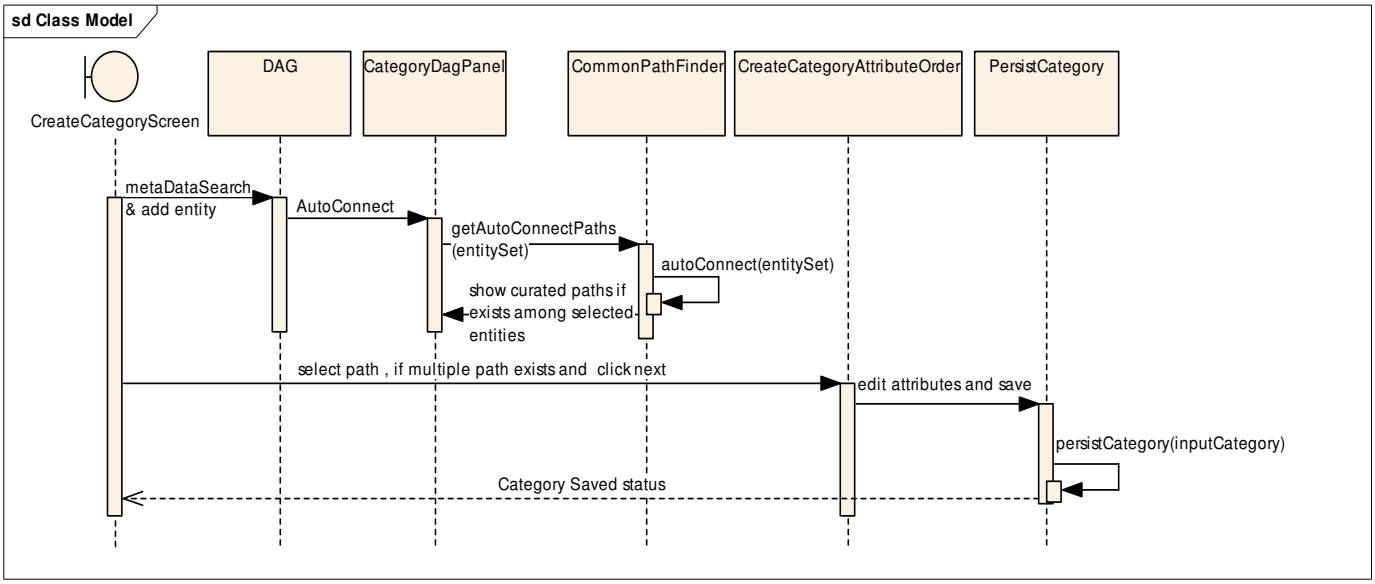

Figure 10 Category creation with Auto Connect

Description:

After adding entities into DAG, existing paths between two selected entities can be shown by either by clicking Manual Connect button or Auto Connect button.

On clicking Manual Connect, DAG.mxml calls *getPaths(soruceEntity,targetEntity)* of *CategoryDagPanel* which subsequently calls the *getAllPath(soruceEntity,targetEntity)* of *CommonPathFinder*. This activity returns the general/curate paths between the source and target entity. If multiple general/curate paths exist between the source and target entity, then Ambiguity Resolver pops up showing all the genera/curated paths.

On clicking AutoConnect button, *DAG,mxml* class calls *getAutoPaths(entitySet)* of *CategoryDagPanel* which calls the *autoConnect(entitySet)* of *CommonPathFinder*. If multiple paths exist among selected entities, then Ambiguity Resolver pops up showing all the paths. Clicking of Next button shows all the attributes for this category in editable form on *CretaeCategoryAttributeOrder.jsp*. Administrator can edit the attributes names here. On clicking Save button, *persistCategory(inputCategory)* of *PersistCategory* will be called which saves the category.

Chapter 7 InterModel Join

Overview

In InterModel Join module, the concept of path curation is extended for defining paths across different models. The matching attribute is used in joining and the classes are end points of that inter-model bridge.

Class Structure

Classes involved in this module are as follows:

- **InterModelConnection**, is the domain object that represents the connection or link between two entities, both from different models.
- **InterModelConnectionsUtil**, contains the business logic that is needed for the creating and persisting the InterModel connection.
- **InterModelMatching**, is the action class that gets two selected entities form UI and determines the possibility of their connection by calling, *InterModelConnectionsUtil.determineConnections(entity1, entity2)* method. If the two entities can be connected then it redirects the result and displays the set of attributes, through which the connection can be established.
- **PersistInterModel**, is the action class that gets the attribute pair from the UI and saves the InterModel connection by calling, *InterModelConnectionsUtil.saveInterModelConnection(attributePair)* method.

Database schema

Save-Update-delete operations will be performed using DAO pattern.

The DDL of the tables will be

```
drop table if exists ASSOCIATION;
drop table if exists INTER_MODEL_ASSOCIATION;

/* Possible values for ASSOCIATION_TYPE are 1 and 2
ASSOCIATION_TYPE = 1 represents INTER_MODEL_ASSOCIATION.
ASSOCIATION_TYPE = 2 represents INTRA_MODEL_ASSOCIATION.
*/
create table ASSOCIATION(
    ASSOCIATION_ID    bigint    not null,
    ASSOCIATION_TYPE  INT(8)    not null ,
    primary key (ASSOCIATION_ID)
);

create table INTER_MODEL_ASSOCIATION(
ASSOCIATION_ID bigint  not null references
ASSOCIATION(ASSOCIATION_ID),
```

```
    LEFT_ENTITY_ID      bigint  not null,
    LEFT_ATTRIBUTE_ID   bigint  not null,
    RIGHT_ENTITY_ID     bigint  not null,
    RIGHT_ATTRIBUTE_ID  bigint  not null,
    primary key (ASSOCIATION_ID)
);
```

Sequence Diagram

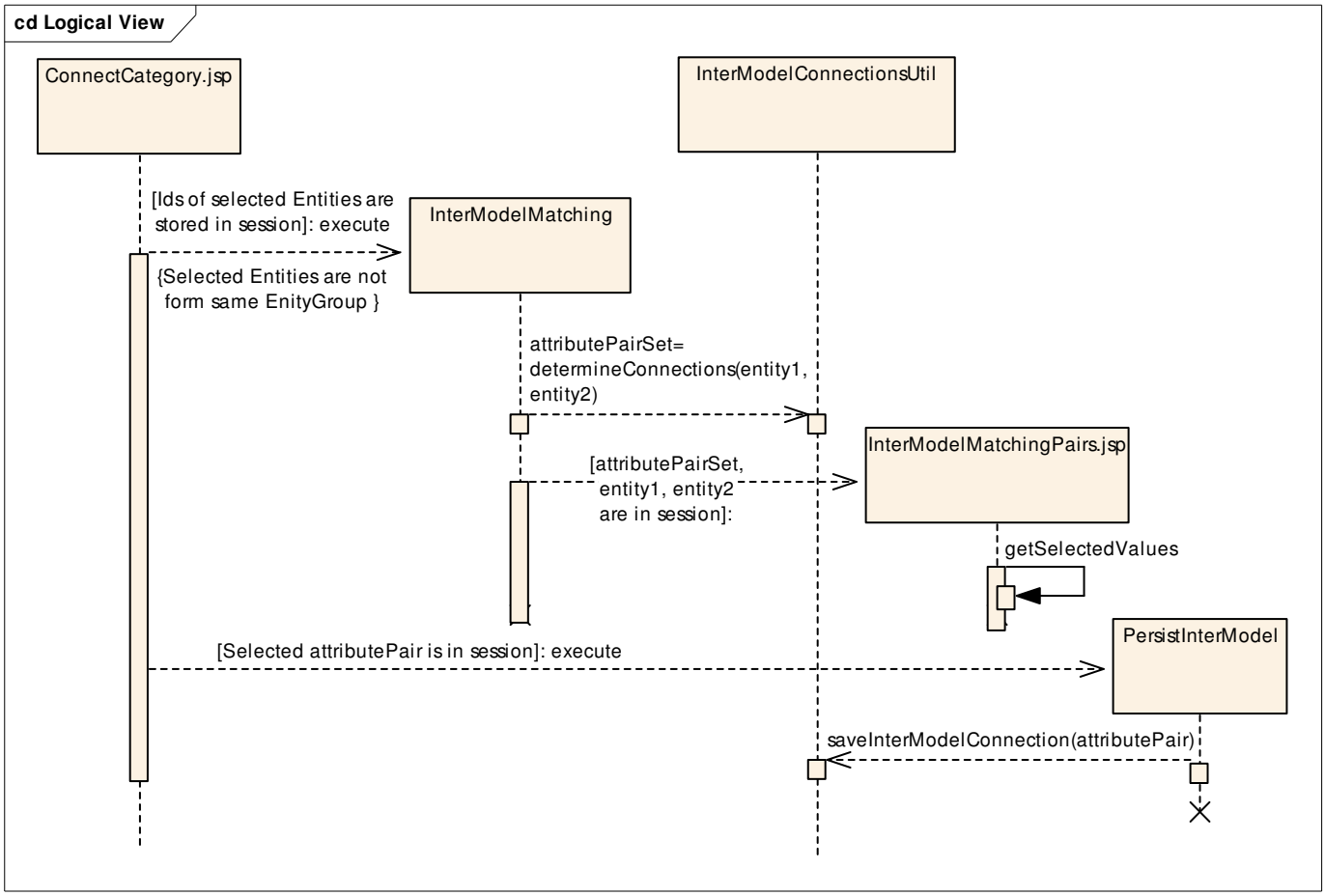

Figure 11 Sequence diagram of InterModel Join

Description:

On ConnectCategory.jsp, user selects the two node added in the DAG. Clicking "Join Classes" button invokes InterModelMaching action. Selected entities are passed to it via session.

InterModelMaching action class in turn invokes the determineConnections() of InterModelConnectionsUtil. This method determines the matching factor between the two selected classes as per the following procedure.

For curating path across two models, the administrator adds two classes, which will be the end points of inter-model bridge, in the DAG. To connect these two ends of the bridge, following process is compiled. In the process, each successive step is reached only if no results are found in the previous step.

Step 1: In models from caDSR, public ID is defined only for attributes. An attribute's public ID is same as the Data Element public ID from the CDE browser. The application shows exact common data element (CDE) matches. e. g. application show attributes which belongs to bridge classes and which have the same public ID.

Step 2: Showing exact data element concept (DEC) match. Ideal way of doing this is matching their public ID. As these public IDs are absent in XML, this is simulated by matching the concept codes of attribute and class (not value domain) in order. If they match, that attribute is eligible for joining.

Step 3: Showing attributes having same concept codes in the same order (i.e. property and property qualifier concept codes are in the same order).

Step 4: Showing all attributes of bridge classes and ask the Administrator to select attributes to be used to join. This is manual join.

The returns set of matching attribute pairs which are displayed on InterModelMatchingPairs.jsp.

On this page user selects the pair to be used as join between the two classes. The pair is reflected in the DAG as a link between the two selected classes.

Clicking Save Join button on ConnectCategory.jsp invokes PersistInterModel action which takes selected attributePair via session.

PersistInterModel action class invokes saveInterModelConnection() of InterModelConnectionsUtil . This creates the join between the inter-model classes.